Catalog Manager

This chapter describes the Catalog Manager, which provides access to AOCE catalogs, including PowerShare server-based catalogs, personal catalogs, and external catalogs. AOCE catalogs are repositories of information that have a standard interface defined by AOCE system software. Although AOCE catalogs are commonly used to store addresses for mail and messaging services, there are no restrictions on the type or internal structure of the data they may contain. This chapter tells you how to use the Catalog Manager to create, modify, and read information in AOCE catalogs.

You can add a catalog-browsing interface to your application with the routines described in the chapter "Standard Catalog Package" in this book. Users can browse and modify the information in AOCE catalogs through the Finder when they install PowerTalk system software. The AOCE Catalogs Extension to the Finder is described in the chapter "AOCE Templates," which also tells you how to extend the catalog browser to handle new types of data.

This chapter describes the nature and types of AOCE catalogs and presents the low-level interface to AOCE catalogs. You can use the routines in this chapter if you want to provide capabilities to access catalogs beyond those provided by the Standard Catalog Package and the catalog browser. If you are creating an AOCE catalog service access module for another type of catalog, you need the information in this chapter plus the chapter "Catalog Service Access Modules," in *Inside Macintosh: AOCE Service Access Modules*.

This chapter starts with a general introduction to AOCE catalogs, followed by an introduction to the Catalog Manager. Then it describes how you can use the Catalog Manager to

- get information about AOCE catalogs and catalog nodes
- create, open, and close personal catalogs
- manage the organization of an AOCE catalog
- manage the content of an AOCE catalog
- control access to a catalog and to the contents of a catalog

Apple's PowerShare serves include a catalog and authentication server. The authentication process determines whether a user should be granted access to a PowerShare catalog. The application program interface (API) for the identification and authentication of users is handled by the Authentication Manager, described in the chapter "Authentication Manager" in this book.

For a general overview of AOCE, see the chapter "Introduction to the Apple Open Collaboration Environment" in this book.

Introduction to AOCE Catalogs

There are three types of AOCE catalogs: PowerShare server-based catalogs, personal catalogs, and external catalogs. You use the same set of Catalog Manager routines to read and modify the contents of any of these catalogs. The term "AOCE catalog" may refer to any or all of these types of catalogs.

A ***PowerShare server*** uses the Apple Catalog and Authentication Protocol to communicate with the AOCE Catalog and Authentication Managers. A PowerShare server can be installed on an AppleTalk network to provide catalog services to any number of entities on that network. In addition to providing a PowerShare catalog, a PowerShare server can identify and authenticate users to ensure that only authorized people or agents gain access to the catalog information. For each user, the server administrator can restrict access to the entire catalog or to any portion of the data in the catalog.

A ***personal catalog*** is an HFS (Hierarchical File System) file located on a user's local disk. A personal catalog can store anything that can be kept in a PowerShare catalog and is often used to store frequently used information from such a catalog.

An ***information card*** is a type of personal catalog that contains a single record. Typically, it contains all of a user's electronic address information. Because it contains only one record, it can be sent quickly and easily to other users as needed.

An ***external catalog*** is one that is accessible to your application through the Catalog Manager API by means of a ***catalog service access module*** (***CSAM***). You access and use an external catalog exactly as you do a PowerShare catalog. The services of the Catalog Manager can be extended to any catalog through a catalog service access module. You do not need to know about catalog service access modules to gain access to external catalogs through the Catalog Manager.

Every catalog provides a set of capability flags that define which features of the Catalog Manager API the catalog supports. In general, your application uses the capability flags to determine what it can do relative to a given catalog; the underlying catalog type (PowerShare, personal, or external) is, with few exceptions, irrelevant. The capability flags are discussed in "Feature Flag Bit Array" beginning on page 8-186. That section also contains information about what features are supported by all PowerShare catalogs and all personal catalogs.

Each catalog is identified by a name and a reference number known as a ***catalog discriminator.*** The combination of name and discriminator is almost certain to be unique, and you must use both when calling Catalog Manager routines to address PowerShare or external catalogs. The Catalog Manager returns a *personal catalog reference number* when you open a personal catalog, and you use that number when addressing a personal catalog through the Catalog Manager API.

Catalog Nodes

AOCE catalogs contain information arranged in a hierarchical structure similar to that of the Macintosh hierarchical file system (HFS). At the root level of the hierarchy is the AOCE catalog itself. Each catalog can contain any number of nodes; a ***catalog node*** (or ***dNode***) is a container that can hold other dNodes, records, or both dNodes and records. A dNode is analogous to an HFS folder, which can contain other folders, files, or both folders and files. A ***record*** is analogous to an HFS file. A record contains the actual data stored in the catalog.

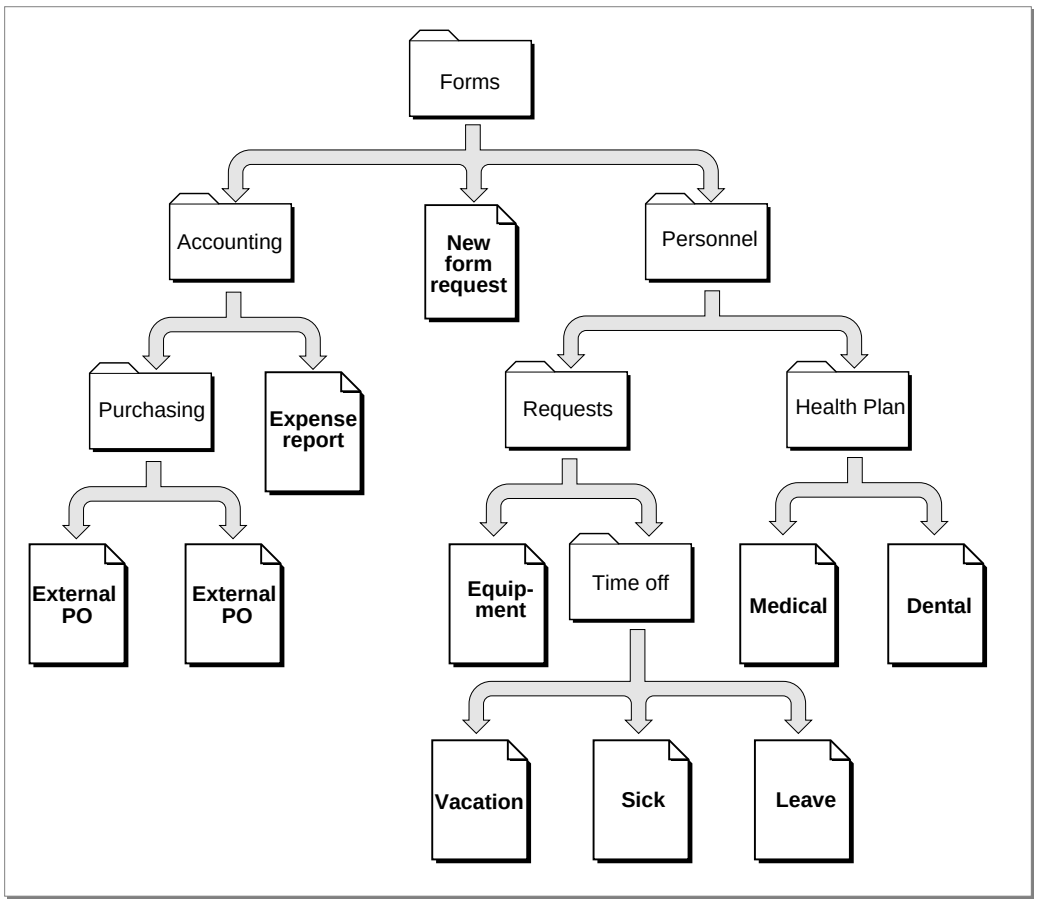
You can identify a specific node within a catalog in three ways. You can specify a dNode by its pathname. A ***pathname*** consists of the name of each dNode in the catalog tree starting from the first node after the root node and including each intervening node to the node in question. In addition, some catalogs assign a unique number, called a ***dNode number,*** to each dNode. For such catalogs, you can use the dNode number rather than the pathname to identify a particular dNode. A partial pathname specification is the third way to identify a dNode. Not all catalogs accept a partial pathname. A ***partial pathname*** consists of a dNode number plus the name of each dNode starting from the one after the dNode specified by the dNode number and continuing to the node in question.

Figure 8-1 on page 8-164 illustrates the structure of a sample AOCE catalog. In this example, the catalog, named Forms, contains personnel and accounting forms for a small company. The stacks of documents in the figure represent catalog nodes, and the individual documents in the figure represent records containing forms. Immediately under the catalog are two nodes, named Accounting and Personnel, and one record containing a form to request new forms. Notice that any given node can contain both records and other nodes. In this example, the pathname for the node containing time-off-request forms is Personnel:Requests:Time off, and the pathname for the node containing purchase orders is Accounting:Purchase. (The colons in the pathname are included for readability only; they are not part of the actual pathname.)

Unlike HFS pathnames, which include the volume name, AOCE pathnames do not include the name of the root catalog. Assume that the nodes named Personnel, Requests, and Time off have the dNode numbers 10, 20, and 30, respectively. In this case, you can either identify the node containing time-off-request forms with a partial pathname that consists of the dNode number 10 and the path Requests:Time off or with the dNode number 20 and the path Time off.

Note that a specific type of catalog might support all or only part of this model. For example, a personal catalog contains only records, no dNodes. An external catalog may support all or any part of this catalog structure.

Figure 8-1 Structure of an AOCE catalog



Catalog Records and Attributes

A record is uniquely identified by a ***record ID*** that allows the Catalog Manager to classify and locate the record. The Catalog Manager defines the structure of a record but places no restrictions on the type of data it may contain.

A record ID consists of

- record location information
- a record name
- the record type
- a creation ID

The record location information consists of the catalog name and discriminator, the dNode number, and the pathname for the dNode containing the record. The record name can be any string of type `RString` (type `RString` is described in the chapter "AOCE Utilities" in this book). The ***record type*** indicates the type of entity represented by the record; for example, Printer, User, or Icon. Apple Computer, Inc., defines certain

record types to facilitate collaboration within the AOCE environment; you can define additional record types. The ***record creation ID***, assigned by the catalog, is a number that uniquely identifies the record within the catalog. Typically, a user interface uses the record name and type to identify the record, whereas software uses the record creation ID. Not all catalogs support record creation IDs. If a catalog does not support record creation IDs, you use the record name and record type to identify a record.

The information in a record is stored in attribute. An ***attribute*** is completely specified by an attribute type, an attribute creation ID, an attribute tag, and the actual attribute value. Attribute values are grouped together by attribute type. The ***attribute type*** reflects the type of data stored in the attribute value; for example, telephone number, mailing address, or picture. Apple defines a number of attribute types; you can define additional attribute types. An attribute type may have zero or more attribute values associated with it. An ***attribute creation ID*** uniquely identifies the attribute value. Some catalogs may not support attribute creation IDs. If a catalog does not support attribute creation IDs, you use the attribute value itself and the attribute type to identify an attribute value. An ***attribute tag*** indicates the format of the attribute value. Apple Computer has defined a few attribute tags for use by Apple's PowerShare catalogs; developers of catalog service access modules can define their own attribute tags to support collaborative applications. There is a maximum size for attribute values stored in PowerShare and personal catalogs, but there are no restrictions on their content. The `Attribute`, `AttributeType` and `AttributeValue` data types are described in the chapter "AOCE Utilities" in this book.

Aliases and Pseudonyms

Some catalogs support the use of alternative names (or ***pseudonyms***) for catalog records. For example, the record "Sally Simon" might have the pseudonym "Simon, Sally". You can use a pseudonym in any Catalog Manager routine that requires a record name.

The Catalog Manager also allows you to create aliases for records. A ***record alias*** is itself a record and so can be placed in any catalog. The Catalog Manager creates the record and marks it as an alias. It is up to you to store the information that your application requires to resolve the alias. For example, you might store the record location information for the original record in an attribute value in the alias record. Not all catalogs support the ability to create aliases.

Access Controls

The Catalog Manager defines access controls for dNodes, records, and attribute types. Types of access privilege include the ability to add and delete attribute types within records, to see attribute values, to add and delete records within dNodes, and so forth. Service requesters are either authenticated, that is, represented by a record in the catalog that contains the dNode, record, or attribute type to which they seek access, or they are guests. There are several categories of authenticated requesters. Access controls are discussed in more detail in the section "Getting Access Controls" on page 8-169.

PowerShare catalogs support a full range of access controls. External catalogs can support any level of access controls up to the full set defined by the Catalog Manager. Personal catalogs do not support Catalog Manager access controls as such. Instead, personal catalogs derive their access controls from the read and read/write privileges allowed by the Macintosh file system. However, you can use the Catalog Manager to read the access controls for a personal catalog. In that case, the Catalog Manager maps file system settings into its own access control settings.

Identities and the PowerTalk Setup Catalog

The PowerTalk system software creates a special personal catalog called the PowerTalk Setup catalog. The ***PowerTalk Setup catalog*** contains information about the catalogs and electronic mail systems that are available to the principal user of the computer. The PowerTalk Setup catalog is a personal catalog stored on the user's local disk. The records in the PowerTalk Setup catalog represent, among other things, PowerShare catalogs, external catalogs, and catalog service access modules. Catalogs and catalog service access modules represented by records in the PowerTalk Setup catalog are said to be "listed in PowerTalk Setup." The Catalog Manager provides routines that allow you to add and remove records from this and other personal catalogs.

Most Catalog Manager routines take an identity as an input. An ***identity*** is a number derived from a user name and password.

A "master" name and password protect the information in the PowerTalk Setup catalog. When a user enters his or her name and password after starting up PowerTalk, the Authentication Manager transforms the name and password into a special value called the local identity. The ***local identity*** is a "master" identity that provides you with transparent access to all of the specific names and passwords stored in the PowerTalk Setup catalog. You can obtain the local identity by calling the Authentication Manager's `AuthGetLocalIdentity` function.

There is another type of identity called specific identity. A ***specific identity*** is derived from the name and password of a user who has an account on a specific server. This user can be the principle user of the computer or an alternate user (or *visitor*). Specific identities make it possible for several people to use the same Macintosh to gain access to their PowerShare catalog services. You can obtain a specific identity by calling the Authentication Manager's `AuthBindIdentity` function.

The identity that you provide is used to determine if the requester is authorized to make the service request. In any Catalog Manager function, you may specify either a local identity, a specific identity, or 0, which indicates guest access. If you specify the local identity, you do not need to know the requester's specific identity for a particular catalog. The Catalog Manager uses the local identity to obtain the specific identity before processing the request.

PowerShare catalogs require an identity for most service requests; external catalogs may not. Personal catalogs do not require an identity with service requests.

For more information about the PowerTalk Setup catalog, see the chapter "Service Access Module Setup" in *Inside Macintosh: AOCE Service Access Modules*. For more

information about local identity and specific identities, see the chapter "Authentication Manager" in this book.

About the Catalog Manager

The Catalog Manager, the Interprogram Messaging Manager, and the Authentication Manager together constitute the fundamental AOCE services. The Catalog Browser and the Standard Catalog Package provide high-level interfaces to the Catalog Manager, and catalog service access modules provide a way for developers to extend AOCE catalog services to external catalogs. See the chapter "Introduction to the Apple Open Collaboration Environment for a description of the position of the Catalog Manager within the AOCE software architecture.

The Catalog Manager includes routines that provide the following services:

- getting information about catalogs, including the catalogs that are listed in the PowerTalk Setup catalog, getting information about a specific catalog's capabilities, getting information about the icons that represent a specific external catalog, and obtaining the name of the network in which a catalog is located
- getting information about a catalog hierarchy, including enumerating dNodes, mapping dNode numbers to pathnames and vice versa, detecting changes in dNodes, and getting information about a specific dNode
- managing the PowerTalk Setup catalog, including listing a PowerShare catalog in PowerTalk Setup, removing a PowerShare catalog from PowerTalk Setup, and searching a network for PowerShare catalogs that you want to use
- creating, opening, and closing personal catalogs
- managing records, including adding and deleting records and aliases, listing records and aliases, getting and setting the name and type of a record, getting information about a record, and adding, deleting, and listing pseudonyms
- managing attribute values and types, including adding and deleting attribute values, changing attribute values, looking for specific attribute values, looking up attribute values, and listing attribute types
- determining access to dNodes, records, and attribute types

Note that the Catalog Manager API does not provide routines for catalog configuration and administration that allow you to create a catalog, to name or rename a catalog, to add and delete nodes, and so forth. These functions, unique to each type of catalog, are handled by the catalog's administration software and are beyond the scope of the Catalog Manager.

Get/Parse Function Pairs

The Catalog Manager API supplies several get/parse function pairs that work together to provide you with information about dNodes, records, access controls, and so forth. The "get" routine of each of these pairs writes the data in a format that is private to the

Catalog Manager into a buffer that you supply. The corresponding "parse" routine extracts the data from the buffer and passes it in logical chunks to a callback routine that you supply.

For example, the `DirEnumerateDirectoriesGet` function stores in a buffer information about all of the catalogs that are listed in the PowerTalk Setup catalog. The `DirEnumerateDirectoriesParse` function parses that information and calls your callback routine for each catalog about which there is information in the buffer. Each time it calls your callback routine, the parse function passes it a catalog name, the catalog discriminator, and information about the features supported by the catalog.

Callback Routines

When you call a Catalog Manager parse function, you pass it a pointer to a callback routine that you provide. If you call the parse function synchronously, the same execution environment (low-memory global variables, A5 world, stack, interrupt state, and any programming restrictions) that was in effect when the Catalog Manager began executing the parse function is also in effect when your callback routine is executed. Therefore, if it is safe to allocate memory or make synchronous calls when you call the parse routine, then your callback routine can also allocate memory or make synchronous calls.

If you call the parse function asynchronously, it saves only the A5 world and restores it when it calls your callback routine. In this case you have access to your application's global variables, but you cannot allocate memory or make synchronous calls.

Callback routines should not call Catalog Manager functions, call the `WaitNextEvent` or `SystemTask` routines, invoke the Notification Manager, or call any function that calls any of these routines.

One of the parameters a parse function passes to your callback routine is the value you placed in the `clientData` field of the parse function's parameter block. You can use this value for whatever purpose you wish; for example, you can use it to distinguish between asynchronous parse requests if you have more than one pending completion or use it to point to a private data area.

The parameters that a parse function passes to a callback routine are described under each routine in the section "Application-Defined Functions" beginning on page 8-308.

Every callback routine returns a Boolean result. If you want the parse function to continue parsing the data in your buffer, return `false`; otherwise, return `true`.

Determining Features Supported

A catalog may not support all the features of the Catalog Manager API. You call the `DirGetDirectoryInfo` function to determine the features that a catalog supports before calling other Catalog Manager functions that address that catalog. The feature information is specified in a feature flag bit array. The bits are defined in "Feature Flag Bit Array" beginning on page 8-186.

Getting Access Controls

The information discussed in this section applies primarily to PowerShare catalogs. Access controls for personal catalogs consist of read and read/write settings implemented by the File Manager. These are mapped into the Catalog Manager access control privileges when you ask for the access controls for a personal catalog.

Three interrelated components to the access controls are available through the Catalog Manager. The first component is the container whose access is controlled. Consider dNodes, records, and attribute types as sets of nested abstract containers. DNodes may contain records, aliases, pseudonyms and other dNodes; records may contain attribute types; and attribute types may contain attribute values. The second component is the requester seeking access to a container. The third component is the kind of access privilege that the requester seeks. DNodes, records, and attribute types are discussed in "Access Controls" on page 8-165. Requesters and access privileges are discussed in the following sections.

Types of Requesters

PowerShare catalogs classify all requesters into five categories. Each dNode, record, and attribute type maintains a set of access privileges for each of the categories. A single requester may fall into one or more categories.

An external catalog, by contrast, does not necessarily use requester categories. It may maintain access privileges for each individual requester. Alternatively, it may use categories different from those used by PowerShare catalogs.

You use a variable of type `CategoryMask` to specify the type of requestor about which you want information.

```
typedef unsigned long CategoryMask;
```

The bits in the `CategoryMask` data type are defined as follows:

```
enum {
   kThisRecordOwnerBit           =  0,
   kFriendsBit                   =  1,
   kAuthenticatedInDNodeBit      =  2,
   kAuthenticatedInDirectoryBit  =  3,
   kGuestBit                     =  4,
   kMeBit                        =  5
};
```

You can use the following values to set and test the bits in a variable of type `CategoryMask`.

```
enum {         /* Values of CategoryMask */
   kThisRecordOwnerMask          = (1L << kThisRecordOwnerBit),
   kFriendsMask                  = (1L << kFriendsBit),
```

```
kAuthenticatedInDNodeMask     = (1L << kAuthenticatedInDNodeBit),
kAuthenticatedInDirectoryMask = (1L << kAuthenticatedInDirectoryBit),
kGuestMask                    = (1L << kGuestBit),
kMeMask                       = (1L << kMeBit)};
```

Descriptions

`kThisRecordOwnerMask`
: A requester in this category is the owner of the record or attribute type to which the requester wants access. (This category has no meaning at the dNode level.) At most, only one requester can belong in this category for each record or attribute type. The owner of a record is the person or process represented by the record. The owner of an attribute type is the person or process represented by the record that contains the attribute type. The creation ID of the requester's own record is the same as the creation ID of the record to which access is sought. (Or the record names and record types are the same in catalogs that do not support creation IDs.)

`kFriendsMask`
: A requester in this category is specially selected and may have different (usually broader) access privileges to a dNode, record, or attribute type than those available to requesters who belong to the more general categories. For PowerShare catalogs, the attribute type `kOwnersAttrTypeNum` is defined to identify persons or processes as friends. An attribute value of attribute type `kOwnersAttrTypeNum` is a `DSSpec` data structure that contains a record ID. Every requester represented by such a value in the `kOwnersAttrTypeNum` attribute type belongs by definition to the friends category. You can add a person or process to the friends category by adding a value specifying that person or process to the `kOwnersAttrTypeNum` attribute type. Note that to do this you need a level of access privilege that allows you to change the access control privileges for a dNode, record, or attribute type as well as to add values. In PowerShare catalogs, the requesters in the friends category for any attribute type within a record are exactly the same as the requesters in the friends category for the record itself.

`kAuthenticatedInDNodeMask`
: A requester in this category is represented by a record located in the same dNode as the dNode, record, or attribute type to which the requester wants access.

`kAuthenticatedInDirectoryMask`
: A requester in this category is represented by a record located in the same catalog as the dNode, record, or attribute type to which the requester wants access.

`kGuestMask`
: A requester in this category is not represented by a record that resides in the same catalog as the dNode, record, or attribute type that the requester wants to access.

`kMeMask`	This is a quasi-category called "me." The `DirGet`*xxx*`AccessControlGet` functions provide it as an output category when the requester asks only for information on his or her own access privileges. It is a convenient way of providing access privilege information that pertains to the requester regardless of the categories to which that requester belongs.

Types of Access Privileges

The Catalog Manager defines different kinds of access privileges. These kinds of privileges are specified by the access control bit masks that are described next.

Mask descriptions

`kNoPrivs`	This mask specifies that the requester has no access to a dNode, a record, or an attribute type.
`kSeeMask`	When the container is a dNode, this mask specifies the ability to view the records, aliases, and pseudonyms of the dNode. When the container is a record, this mask specifies the ability to view the contents of the record. When the container is an attribute type, this mask specifies the ability to view the contents of the attribute type.
`kAddMask`	When the container is a dNode, this mask specifies the ability to add records, aliases, pseudonyms, and dNodes to the dNode. When the container is a record, this mask specifies the ability to add attribute types to the record. When the container is an attribute type, this mask specifies the ability to add values to the attribute type.
`kDeleteMask`	When the container is a dNode, this mask specifies the ability to delete records, aliases, pseudonyms, and dNodes from the dNode. When the container is a record, this mask specifies the ability to delete attribute types from the record. When the container is an attribute type, this mask specifies the ability to delete values from an attribute type.
`kChangeMask`	When the container is a record, this mask specifies the ability to change the contents of the record; that is, to replace some or all of the record's contents without changing the record's creation ID. When the container is an attribute type, this mask specifies the ability to change the contents of an attribute type; that is, to replace an attribute value without changing the attribute creation ID. Changing the contents of a dNode is undefined and not supported.
`kRenameMask`	When the container is a dNode or a record, this mask specifies the ability to rename it by changing its record name and record type. Renaming an attribute type is not supported.

`kChangePrivsMask`
: This mask specifies the ability to change the access control privileges for a dNode, a record, or an attribute type. The ability to change access privileges for a container includes the ability to change privileges for the content of the container as well. If you can change access privileges for a dNode, you can also change access privileges for the records and attribute types within the dNode. Likewise, if you can change access privileges for a record, you can also change access privileges for the attribute types within the record.

`kSeeFoldersMask`
: When the container is a dNode or a record, this mask specifies the ability to view dNodes within the container dNode.

`kAllPrivs`
: This mask specifies the sum of all the specific access privileges just described for a dNode, a record, or an attribute type.

Access Control Lists

PowerShare catalogs maintain an access control list for each dNode, record, and attribute type in the catalog. Each entry in an access control list specifies a category and the category's access privileges with respect to that dNode, record, or attribute type. Each access control list consists of five entries, one for each category.

The Catalog Manager API, however, does not restrict access control lists to the PowerShare implementation. An external catalog may maintain access control lists that consist of individual requesters and their access privileges, instead of categories of requesters. Or it may consist of the PowerShare categories or different categories or some combination of these. The access control list of an external catalog may have any number of entries.

A personal catalog has a single entry in its access control list to which every requester belongs. For a personal catalog, every requester has exactly the same access privileges.

Regardless of the type of access control list that a catalog maintains, all catalogs should be able to provide a requester with the access privileges that apply to that requester. (This is the quasi-category "me" for PowerShare catalogs.)

PowerShare catalogs implement access controls for most catalog service requests. The PowerShare catalog identifies the categories to which the requester belongs and determines if a requester in those categories has sufficient access to perform the requested action.

The Catalog Manager provides a get/parse pair of functions for each type of container so that you can obtain access control information about the container. A `DirGet`*xxx*`AccessControlGet` function obtains access control information from the access control list associated with a dNode, record, or attribute type and stores the information in a buffer that you provide. A `DirGet`*xxx*`AccessControlParse` function retrieves information for one access control list entry at a time from your buffer and passes it to your callback routine. You can request access control information for every entry on the access control list, for a subset of entries, or for only the requester (specified in the `identity` field of the `DirParamBlock` parameter block).

Note
The utility routine `OCEGetAccessControlDSSpec` converts a category mask into the catalog service specification that you need to pass to a `DirGet`*xxx*`AccessControlGet` function. This routine is described on page 8-290. ◆

For PowerShare catalogs and personal catalogs, the information that a `DirGet`*xxx*`AccessControlParse` function passes to your callback routine consists of a catalog service specification that identifies a category plus the access control masks that apply to that category. The access control masks may be described as either active or default. An active access control mask is a mask that currently applies to a dNode, record, or attribute type; it specifies which operations a requester in the category is authorized to perform on the dNode, record, or attribute type. A default access control mask is the mask that is applied to new objects within the container at the time they are created. For example, any new records that are created within a dNode automatically acquire the access controls that are specified by the dNode's default access control mask for records.

When you request dNode access control information, you get the active access control mask for the dNode as well as the default access control masks that apply to newly created records and attribute types within the dNode. When you request record access control information, you get the active access control masks for the record and the dNode containing the record. You also get the default access control mask that applies to newly created attribute types within the record. When you request attribute type access control information, you get the active access control masks for the attribute type as well as the record and the dNode containing the attribute type.

Using the Catalog Manager

The Catalog Manager API supplies several get/parse function pairs that work together to provide you with information about dNodes, records, access controls, and so forth. The "get" routine of each of these pairs writes the data in a format that is private to the Catalog Manager into a buffer that you supply. The corresponding "parse" routine extracts the data from the buffer and passes it in logical chunks to a callback routine that you supply.

If the initial buffer size is not sufficient to hold all the data, there are two different ways in which the get/parse function pairs work—depending on which Catalog Manager routines are called. The first example, "Getting Attribute Value Information" beginning on page 8-174 shows how the `DirLookupGet` and `DirLookupParse` work together to extract information. These two functions use identical parameter blocks They are the only get/parse function pair that work this way.

The example, "Getting Attribute Type Information" beginning on page 8-178 illustrates how all other get/parse function pairs work.

There is also one "get" function, `DirGetExtendedDirectoryInfo`, that has no corresponding "parse" routine. In this case, you must write your own "parse" routine. See "Getting Extended Catalog Information" beginning on page 8-182 for an example that shows how to do this.

Determining Whether the Collaboration Toolbox Is Available

Before calling any of the Catalog Manager functions, you should verify that the Collaboration toolbox is available by calling the `Gestalt` function with the selector `gestaltOCEToolboxAttr`. If the Collaboration toolbox is present but not running (for example, if the user deactivated it from the PowerTalk Setup control panel), the `Gestalt` function sets the bit `gestaltOCETBPresent` in the `response` parameter. If the Collaboration toolbox is running and available, the function sets the bit `gestaltOCETBAvailable` in the `response` parameter. The Gestalt Manager is described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities*.

If you want to be informed when the Catalog Manager starts up or shuts down, you can install an entry in the AppleTalk Transition Queue (ATQ). Then the AppleTalk LAP Manager calls your ATQ routine with the transition selector `ATTransDirStart` when the Catalog Manager has finished starting up and with the selector `ATTransDirShutdown` when the Catalog Manager has started to shut down. The ATQ is described in the chapter "Link-Access Protocol (LAP) Manager" in *Inside Macintosh: Networking*.

Determining the Version of the Catalog Manager

To determine the version of the Catalog Manager that is available, call the `Gestalt` function with the selector `gestaltOCEToolboxVersion`. The function returns the version number of the Collaboration toolbox in the low-order word of the `response` parameter. For example, a value of 0x0101 indicates version 1.0.1. If you are using the Collaboration toolbox on a computer that has a PowerShare server, the function returns the version number of the server in the high-order word of the `response` parameter. If the Collaboration toolbox or server is not present and available, the `Gestalt` function returns 0 for the relevant version number. You can use the constant `gestaltOCETB` for AOCE Collaboration toolbox version 1.0.

Getting Attribute Value Information

The `DoProcessAttributeValues` function in Listing 8-1 lists the attribute values for a particular catalog. It uses two Catalog Manager routines: `DirLookupGet` (page 8-276) and `DirLookupParse` (page 8-279). Because these two routines have the same parameter block, you can call `DirLookupGet` as many times as necessary, using the same parameter block you last passed to `DirLookupParse`, if the buffer you allocate is too small to hold all the data that `DirLookupGet` returns. Listing 8-2 on page 8-179 shows how to use a pair of get/parse functions that have different parameter blocks.

When `DoProcessAttributeValues` is called, it is passed two parameters, a pointer to the catalog browser and a pointer to the current attribute type. It then calls the `DoEnumerateAttributeValues` function, passing it parameters to specify the identification of the requester; to identify the current catalog, record and attribute type; to name a callback routine; and to match calls to this routine to particular calls to the `DirLookupParse` function.

When `DoEnumerateAttributeValues` returns, `DoProcessAttributeValues` calls `DoSelectNthAttributeValue`—which is not shown here—to extract the current attribute value and display its contents.

The `DoEnumerateAttributeValues` function sets up the parameter block that `DirLookupGet` uses and then sets the initial values. It includes a do/while loop in which `DirLookupGet` and `DirLookupParse` do the work of extracting the attribute value information. If the buffer is too small to hold all the data, the `DirLookupParse` function returns `kOCEMoreData`, and the loop repeats. The same parameter block that was passed to `DirLookupParse` is passed on subsequent calls to `DirLookupGet`. The reason this works is that when `DirLookupParse` completes, it returns values in the `lastRecordIndex`, `lastAttributeIndex`, and `lastAttribute` fields that are at the same offsets in the parameter block as the values of the `startingRecordIndex`, `startingAttrTypeIndex`, and `startingAttribute` fields on a subsequent call to the `DirLookupGet` function. The `DirLookupGet` function continues retrieving information from the point at which it stopped during its previous invocation.

The `DirLookupParse` function calls the callback routine `MyForEachAttrValue` for each attribute value that it finds. The callback routine calls the `DoAddAttribueValue` function—which is not shown here—passing it the data structure containing the attribute values. The `DoAddAttribueValue` function stores the values.

Listing 8-1 Listing the attribute values for a catalog

```
/* Enumerate all attribute types for the currently-selected
   catalog. Attribute types are added to the type list as
   they are found. */

static pascal Boolean      MyForEachAttrValue(
      long                    clientData,
      const Attribute         *theAttribute
   );

/* DoProcessAttributeValues is called when a new attribute type
   is selected.

   Globals
   DOC.currentDsRefNum
        Current personal directory RefNum
   DOC.currentRecordID
```

```
         Current record to examine */

void
DoProcessAttributeValues(
      register CatalogBrowserPtr       dbp,
      const AttributeTypePtr           attributeTypePtr
   )
{
      OSErr                            status;

       /* Make sure to start with a clean slate.*/

      ClearAttributeValueList(dbp);
      status = DoEnumerateAttributeValues(
               DOC.userIdentity,
               DOC.currentDsRefNum,
               &DOC.currentRecordID,
               attributeTypePtr,
               MyForEachAttrValue,
               (long) dbp
            );
      LOG(status, "\pDoEnumerateAttributeValues");
      DoSelectNthAttributeValue(dbp, 0);
}

/* MyForEachAttrValue is called by the DirLookupParse function.
   The attribute value is an RString that is put into the list. */

static pascal Boolean
MyForEachAttrValue(
      long                             clientData,
      const Attribute                  *theAttribute
   )
{
      register CatalogBrowserPtr       dbp;
      Boolean                          stopParse;

      dbp = (CatalogBrowserPtr) clientData;
      stopParse = FALSE;
      TRY {
         AddAttributeValue(dbp, theAttribute);
      }
      CATCH {
```

```
         LOG(STATUS, "\pCan't add attribute value");
         NO_PROPAGATE;
         stopParse = TRUE;
      }
      ENDTRY;
      return (stopParse);
}

#ifndef kMyBufferSize
#define kMyBufferSize 4096
#endif

/* DoEnumerateAttributeValues is called when a new attribute type
   is selected. */

OSErr
DoEnumerateAttributeValues(
      AuthIdentity              userIdentity,
      short                     dsRefNum,
      RecordIDPtr               recordIDPtr,
      const AttributeTypePtr    theAttributeType,
      ForEachAttrValue          MyForEachAttrValue,
      long                      clientData
   )
{
      OSErr                     status;
      AttributeTypePtr          attrTypeList[1];
      RecordIDPtr               recordIDList[1];
      DirParamBlock             dirParamBlock;
      Ptr                       myBuffer;
#define GET    (dirParamBlock.lookupGetPB)
#define PARSE  (dirParamBlock.lookupParsePB)

      myBuffer = NewPtr(kMyBufferSize);
      if (myBuffer == NULL)
         status = MemError();
      else {
         CLEAR(dirParamBlock);
         recordIDList[0] = recordIDPtr;
         attrTypeList[0] = theAttributeType;
         GET.identity = userIdentity;
         GET.ioCompletion = NULL;
         GET.dsRefNum = dsRefNum;
         GET.clientData = clientData;
```

```
            GET.aRecordList = recordIDList;
            GET.attrTypeList = attrTypeList;
            GET.recordIDCount = 1;
            GET.attrTypeCount = 1;
            GET.includeStartingPoint = FALSE;
            GET.getBuffer = myBuffer;
            GET.getBufferSize = kMyBufferSize;
            GET.startingRecordIndex = 1;
            GET.startingAttrTypeIndex = 1;
            CLEAR(GET.startingAttribute);
            do {
               status = DirLookupGet(&dirParamBlock, SYNC);
               if (status == noErr || status == kOCEMoreData) {
                  PARSE.eachRecordID = NULL;
                  PARSE.eachAttrType = NULL;
                  PARSE.eachAttrValue = MyForEachAttrValue;
                  status = DirLookupParse(&dirParamBlock, SYNC);
               }
            } while (status == kOCEMoreData);
            DisposePtr(myBuffer);
         }
         return (status);
#undef GET
#undef PARSE
}
```

Getting Attribute Type Information

The routines in Listing 8-2 return the attribute types for a specified catalog. They use the Catalog Manager `DirEnumerateAttributeTypesGet` (page 8-285) and `DirEnumerateAttributeTypesParse` (page 8-288) functions. As the example shows, if the buffer is too small to hold all the data returned by `DirEnumerateAttributeTypesGet`, it can be called again in a loop, using the last attribute type parameter that `DirEnumerateAttributeTypesParse` passed to the callback routine. Listing 8-1 on page 8-175 shows how a different get/parse pair work together.

The structure `CallBackData` is used to hold data including the current attribute type.

The `DoEnumerateAttributeTypes` function is called by a higher-level routine and is passed parameters to authenticate the user; identify the catalog, the record, and the current attribute; and to match calls to this routine to particular calls to the `DirEnumerateAttributeTypesParse` function. It then allocates a buffer and sets up the parameter block with initial values.

The `DoEnumerateAttributeTypes` function contains a do/while loop that enables `DirEnumerateAttributeTypesGet` and `DirEnumerateAttributeTypesParse` to extract the attribute type information. For each attribute type extracted, `DirEnumerateAttributeTypesParse` calls the `MyEnumerateEachAttributeType` callback routine.

If the buffer is too small to hold all the information returned by `DirEnumerateAttributeTypesGet`, the loop repeats. The `DirEnumerateAttributeTypesGet` function uses as its starting attribute type, the last attribute type that `DirEnumerateAttributeTypesParse` passed to the callback routine.

The callback routine, `MyEnumerateEachAttributeType`, provides a data type to store the attribute type information extracted by `DirEnumerateAttributeTypesParse`. It also stores the last attribute type that it received from `DirEnumerateAttributeTypesParse` in case it needs to pass it back to `DirEnumerateAttributeTypesGet` for another run through the loop.

Listing 8-2 Listing the attribute types for a catalog

```
 /* Enumerate all attribute types for the specified catalog. The caller
    provides a callback function (which takes the same parameters as the AOCE
    DirEnumerateAttributeTypesParse function) that is called with each
    returned attribute type. */

#ifndef kMyBufferSize
#define kMyBufferSize4096
#endif

static pascal Boolean              MyEnumerateEachAttributeType(
      long                            clientData,
      const AttributeType             *aType
   );

/* This data is passed to MyEnumerateEachAttributeType */
 typedef struct CallBackData {
   ForEachAttrType                 eachAttrType;
   long                            clientData;
   AttributeType                   currentAttrType;
} CallBackData, *CallBackDataPtr;

/* DoEnumerateAttributeTypes is called when a new record is selected. It
   calls a user function for each attribute type stored in that record. */

OSErr
```

```
DoEnumerateAttributeTypes(
      AuthIdentity                  userIdentity,
      short                         dsRefNum,
      RecordIDPtr                   recordIDPtr,
      ForEachAttrType               eachAttrType,
      long                          clientData
   )
{
      OSErr                         status;
      Boolean                       first;
      CallBackData                  callBackData;
      DirParamBlock                 dirParamBlock;
      Ptr                           myBuffer;
#define GET    (dirParamBlock.enumerateAttributeTypesGetPB)
#define PARSE  (dirParamBlock.enumerateAttributeTypesParsePB)

      myBuffer = NewPtr(kMyBufferSize);
      if (myBuffer == NULL)
         status = MemError();
      else {
         callBackData.eachAttrType = eachAttrType;
         callBackData.clientData = clientData;
         CLEAR(callBackData.currentAttrType);
         CLEAR(dirParamBlock);
         first = TRUE;
         do {
            GET.identity = userIdentity;
            GET.dsRefNum = dsRefNum;
            GET.clientData = (long) &callBackData;
            GET.aRecord = recordIDPtr;
            if (first) {
               GET.startingAttrType = NULL;
               first = FALSE;
            }
            else {
               /* This is the last attribute type that was fetched
                  by the parser callback. */
               GET.startingAttrType = &callBackData.currentAttrType;
            }
            GET.includeStartingPoint = FALSE;
            GET.getBuffer = myBuffer;
            GET.getBufferSize = kMyBufferSize;
            status = DirEnumerateAttributeTypesGet(&dirParamBlock, SYNC);
```

```
            if (status != kOCEMoreData)
               LOG(status, "\pDirEnumerateAttributeTypesGet");
            if (status == noErr || status == kOCEMoreData) {

                /* There is a record, or there is a record and more
                   data to read. Parse the data: this will call
                   the callback function.*/
               PARSE.eachAttrType = MyEnumerateEachAttributeType;
               status = DirEnumerateAttributeTypesParse(&dirParamBlock,
                        SYNC);
               if (status != kOCEMoreData)
                  LOG(status, "\pDirEnumerateAttributeTypesParse");
            }
         } while (status == kOCEMoreData);
         DisposePtr(myBuffer);
      }
      return (status);
#undef GET
#undef PARSE
}

/* MyEnumerateEachAttributeType is called by the
   DirEnumerateAttributeTypesParse function. Remember the attribute type for
   the next call and call the application handler. */

static pascal Boolean
MyEnumerateEachAttributeType(
      register long                    clientData,
      const AttributeType              *aType
   )
{
      Boolean                             stopParse;
#define CALLBACK(*((CallBackDataPtr) clientData))

       /* Grab a copy of the attribute type for the next "get more
          data" call. */
      OCECopyRString(
         (const RStringPtr) aType,
         (RStringPtr) &CALLBACK.currentAttrType,
         kAttributeTypeMaxBytes
      );
```

```
    stopParse = CALLBACK.eachAttrType(CALLBACK.clientData, aType);
    return (stopParse);
}
```

Getting Extended Catalog Information

The `DirGetExtendedDirectoriesInfo` function (page 8-212) returns extended information about a catalog. The `DirGetExtendedDirectoriesInfo` function returns a packed structure that you must unpack. The sample routines in Listing 8-3 show how to call the `DirGetExtendedDirectoriesInfo` function and how to examine the information it returns.

The sample routines make use of the structure type, `MyExtendedInfoType`, which can hold the extended information for a single catalog.

The `DoProcessExtendedCatalogInfo` routine declares two pointers: `myBufferPtr`, the pointer to the data buffer that will be passed to the `DirGetExtendedDirectoriesInfo` function, and `extendedInfoPtr`, a pointer to an extended information structure (`MyExtendedInfo`). It sets both pointers to `nil`.

Next, the function calls the `DoGetExtendedCatalogInfo` routine to get the extended information from the Catalog Manager. If the routine returns successfully, `DoProcessExtendedCatalogInfo` allocates enough memory to store the extended information for all of the catalogs on which the Catalog Manager has returned information.

Then `DoProcessExtendedCatalogInfo` calls the `DoUnpackExtendedCatalogInfo` routine to extract the extended information from the data buffer and put it in the array of extended information structures. The `DoUtilizeExtendedCatalogInformation` routine, not shown here, acts on the extended information. You would include a similar routine to do whatever is appropriate to your application. Finally, `DoProcessExtendedCatalogInfo` disposes of the memory that has been allocated before it returns.

Unlike some Catalog Manager functions, `DirGetExtendedDirectoriesInfo` cannot be called to retrieve a portion of the information and then called again to retrieve more. It always attempts to return all of the information at once, and it completes with the `kOCEMoreData` result code if the buffer you pass is too small. The purpose of the `DoGetExtendedCatalogInfo` routine is to call the `DirGetExtendedDirectoriesInfo` function with a buffer that is large enough to hold all the information that `DirGetExtendedDirectoriesInfo` will return. The `DoGetExtendedCatalogInfo` routine allocates a buffer of `kWorkBufferSize` bytes and then calls the `DirGetExtendedDirectoriesInfo` function. If `DirGetExtendedDirectoriesInfo` function returns `kOCEMoreData`, `DoGetExtendedCatalogInfo` disposes of the buffer and allocates a larger one. It does this repeatedly, increasing the size of the buffer in increments of `kWorkBufferSize` bytes until the buffer is large enough to contain all the information `DirGetExtendedDirectoriesInfo` can return. If `DirGetExtendedDirectoriesInfo` completes successfully,

DoGetExtendedCatalogInfo passes back via its actualEntriesPtr parameter the number of catalogs for which extended information now exists in the buffer. Otherwise, it disposes of the buffer.

The DoUnpackExtendedCatalogInfo routine extracts the extended information about each catalog from the buffer and stores the information in MyExtendedInfo structures. It does the same thing as the "parse" routines in Listing 8-1 on page 8-175 and Listing 8-2 on page 8-179, but you have to write this routine yourself because DirGetExtendedDirectoriesInfo has no corresponding "Parse" routine. Note that variables in an extended information structure point to data in the packed buffer.

Because the data in the buffer is of variable length, the sizeof function is required to determine the length of the data. The INCR macro aligns the data on a word boundary.

Listing 8-3 Getting extended information for a catalog

```
typedef struct MyExtendedInfo {
   PackedRLIPtr          pRLIPtr;      /* Catalog's packed RLI     */
   unsigned short        pRLILength;   /* Length of packed RLI     */
   OSType                entnType;     /* Catalog address type     */
   long                  hasMailSlot;  /* Nonzero if mail slot    */
   RStringPtr            realName;     /* Catalog's true name      */
   RStringPtr            comment;      /* More info for display    */
   long                  dataLength;   /* Additional data length   */
   Ptr                   dataPtr;      /* Additional information   */
} MyExtendedInfoType, *MyExtendedInfoPtrType;

OSErr DoProcessExtendedCatalogInfo(void) {
   OSErr                   status;
   Ptr                     myBufferPtr;
   MyExtendedInfoPtrType   extendedInfoPtr;
   unsigned long           actualEntries;

   myBufferPtr = nil;
   extendedInfoPtr = nil;
   status = DoGetExtendedCatalogInfo(&myBufferPtr, &actualEntries);
   if (status == noErr) {
      extendedInfoPtr = (MyExtendedInfoPtrType) NewPtrClear(actualEntries *
                        sizeof(MyExtendedInfo));
      status = MemError();
   }
   if (status == noErr) {
      status = DoUnpackExtendedCatalogInfo(myBufferPtr, extendedInfoPtr,
                                           actualEntries);
```

```
      status = DoUtilizeExtendedCatalogInformation(extendedInfoPtr,
                                                   actualEntries);
   }

   if (extendedInfoPtr != nil)
      DisposePtr((Ptr) extendedInfoPtr);
   if (myBufferPtr != nil)
      DisposePtr((Ptr) myBufferPtr);
   return (status);
}
OSErr DoGetExtendedCatalogInfo(
   Ptr               *resultBuffer,       /* address of ptr to buffer */
   unsigned long     *actualEntriesPtr)

{
#define kWorkBufferSize (512)
   OSErr             status;
   DirParamBlock     myParamBlock;
   unsigned long     bufferLength;
#define GET    (myParamBlock.getExtendedDirectoriesInfoPB)
   bufferLength = 0;
   *resultBuffer = nil;
   do {
      if (*resultBuffer != nil)
         DisposePtr(*resultBuffer);
      bufferLength += kWorkBufferSize;
      *resultBuffer = NewPtr(bufferLength);
      if ((status = MemError()) != noErr)
         break;
      ClearMemory(&myParamBlock, sizeof myParamBlock);
      GET.identity = gUserIdentity;
      GET.buffer = *resultBuffer;
      GET.bufferSize = bufferLength;
      status = DirGetExtendedDirectoriesInfo(&myParamBlock, false);
   } while (status == kOCEMoreData);
   if (status == noErr)
      *actualEntriesPtr = GET.actualEntries;
   else if (*resultBuffer != nil) {
      DisposePtr(*resultBuffer);
      *resultBuffer = nil;
   }
   return (status);
#undef GET
}
```

```
OSErr DoUnpackExtendedCatalogInfo(
   register Ptr                     bufPtr,           /* pointer to buffer
                                                         containing packed
                                                         extended catalog
                                                         information */
   register MyExtendedInfoPtrType   extendedInfoPtr,  /* pointer to array
                                            of MyExtendedInfo structures */
   unsigned long                    actualEntries)
{
   unsigned long  dataLength;       /* working value  */
   unsigned long  i;                /* current entry count */

      /* Scan through the buffer to extract the extended catalog
         information. The INCR macro increments bufPtr by some amount,
         making sure that it is aligned on a word boundary. Its argument
         must not have side-effects.*/
#define INCR(v)   (bufPtr += ((v) + ((v) & 0x01)))
#define RESULT    (*extendedInfoPtr)

   for (i = 0; i < actualEntries; i++, extendedInfoPtr++) {
      RESULT.pRLIPtr = (PackedRLIPtr) bufPtr;
      RESULT.pRLILength = pRLIPtr->dataLength;
      INCR(pRLILength + sizeof (ProtoPackedRLI));
      RESULT.entnType = *((OSType *) bufPtr);
      INCR(sizeof (OSType));
      RESULT.hasMailSlot = *((long *) bufPtr);
      INCR(sizeof (long));
      RESULT.realName = (RStringPtr) bufPtr;
      dataLength = RESULT.realName->dataLength;
      INCR(dataLength + sizeof (ProtoRString));
      RESULT.comment = (RStringPtr) bufPtr;
      dataLength = RESULT.comment->dataLength;
      INCR(dataLength + sizeof (ProtoRString));
      RESULT.dataLength = *((long *) bufPtr);
      INCR(sizeof (long));
      RESULT.dataPtr = (Ptr) bufPtr;
      INCR(RESULT.dataLength);          /* Step over the rest */
   }
#undef INCR
#undef RESULT
}
```